SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Ivo Doko

ALGORITMI ZA ODREĐIVANJE NAJBLIŽEG PARA

Diplomski rad

Voditelj rada:
doc. dr. sc. Goranka Nogo

Zagreb, rujan 2014.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Ivici

Sadržaj

Sadržaj **iv**

Uvod **1**

1 Uvodne napomene **2**

2 Početna razmatranja **5**
2.1 Jednodimenzionalni slučaj 5
2.2 Dvodimenzionalni slučaj 6

3 Divide-and-conquer **8**
3.1 Prvi pristup 8
3.2 Poboljšanje složenosti 13

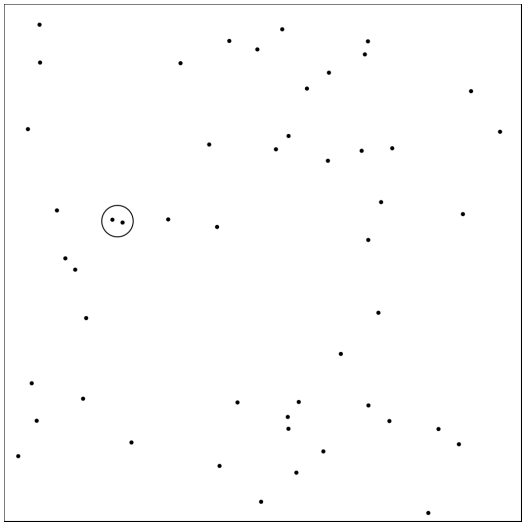
4 Slučajno uzorkovanje **20**
4.1 Rabinov algoritam 20
4.2 Dietzfelbinger-Hagerup-Katajainen-Penttonen algoritam 26

5 Heuristički pristup **31**

Bibliografija **35**

Uvod

Razmatramo sljedeći problem: Za zadani konačni (multi)skup točaka $S \subset \mathbb{R}^m$ potrebno je pronaći $\{a, b\} \in S$ t.d. $d(a, b) = \min_{\{x,y\} \subseteq S} d(x, y)$, tj. najbliži par točaka u S po nekoj metrici d.



Slika 0.1: Primjer skupa točaka u dvije dimenzije. Najbliži par je zaokružen.

Ovaj problem se pojavljuje u području umjetne inteligencije (posebice u razvoju računalnog vida) te u nekim drugim specijaliziranim primjenama kao npr. kontrola leta, gdje je bitno u što manjem vremenskom intervalu za danu grupu zrakoplovā odrediti koja dva su međusobno najbliža (u svrhu sprječavanja sudara).

Uzimat ćemo u obzir samo euklidsku metriku, a cilj nam je proučiti algoritme koji su razvijeni za rješavanje ovog problema za $m = 2$ te usporediti njihove složenosti i realnu efikasnost.[1]

[1] "Realna efikasnost" znači mjerenje vremena potrebnog da bi algoritam riješio problem za različite veličine skupa S.

Poglavlje 1

Uvodne napomene

Kodovi su pisani u jeziku C++, u standardu C++11. U verziji GCC-a koja je bila dostupna za vrijeme pisanja ovog rada (4.9.0), za kompajliranje je bilo potrebno koristiti komandno-linijsku opciju "`-std=c++11`".

Programi su koristili jednu zaglavnu datoteku u kojoj su se nalazile definicije tipova podataka i funkcijā koje su svi programi koristili, kao i ostale potrebne zaglavne datoteke iz STL-a. Pošto je potrebno znati definirane tipove podataka i funkcije da bi se razumjelo kodove u radu, kôd ove datoteke je ovdje prikazan u cijelosti.

```
1  #include <iostream>
2  #include <random>
3  #include <vector>
4  #include <utility>
5  #include <algorithm>
6  #include <unordered_map>
7  #include <cmath>
8  #include <cstdint>
9  #include "seed.hpp"
10
11 using namespace std;
12
13 #define point pair<double, double>
14
15 static mt19937_64 randgen;
16
17 inline double distance(const point& a, const point& b) {
18     double dx = a.first - b.first, dy = a.second - b.second;
19     return (dx*dx + dy*dy);
20 }
21
22 struct result {
23     point a;
```

```
24      point b;
25      double d;
26
27      result() { }
28
29      result(const point& _a, const point& _b, const double& _d)
30          : a(_a), b(_b), d(_d) { }
31
32      result(const point& _a, const point& _b)
33          : a(_a), b(_b), d(distance(_a, _b)) { }
34
35      result(result&& other)
36          : a(move(other.a)), b(move(other.b)), d(move(other.d)) { }
37
38      result(const result& other)
39          : a(other.a), b(other.b), d(other.d) { }
40
41      result& operator= (const result& other) {
42          a = other.a;
43          b = other.b;
44          d = other.d;
45          return *this;
46      }
47  };
48
49  inline bool compare_x(const point& a, const point& b) {
50      return (a.first < b.first);
51  }
52
53  inline bool compare_y(const point& a, const point& b) {
54      return (a.second < b.second);
55  }
56
57  void generate_points(vector<point>& vec) {
58      static uniform_real_distribution<double> dist(0.0, 100.0);
59      for(auto& p : vec) {
60          p.first = dist(randgen);
61          p.second = dist(randgen);
62      }
63  }
```

Za generiranje slučajnih brojeva je korišten `mt19937_64` (Mersenne twister) iz STL `random`. Datoteka "seed.hpp" sadrži samo pomoćne funkcije koje se koriste za postavljanje početnog stanja generatora slučajnih brojeva.[1]

[1] Na POSIX kompatibilnim sistemima se koristi `/dev/urandom`, a na Windows sistemima `RtlGenRandom` iz ADVAPI32.dll.

Mjerenja su obavljena na računalu s procesorom AMD Phenom II X4 955, takta 3.2 GHz. Za mjerenje vremena je korišten IDE Code::Blocks, koji nakon izvršavanja programa ispisuje vrijeme trajanja s rezolucijom tisućinke sekunde. Za procjenu trajanja jednog izvršavanja, algoritmi su izvršavani više tisuća puta u petlji na vektorima odgovarajućih duljinā s nasumično generiranim vrijednostima pri svakom koraku petlje. Kako vrijeme potrošeno na popunjavanje vektora slučajnim vrijednostima ne bi utjecalo na procjenu trajanja algoritama, trajanje generiranja slučajnih vektora odgovarajuće duljine je posebno izmjereno te je naknadno oduzeto od izmjerenog vremena trajanja generiranja vrijednosti i izvršavanja algoritama.

Grafovi su rađeni u programu Wolfram Mathematica 8.0. Na grafovima x-os predstavlja veličinu ulaznih podataka, a y-os procjenu vremena izvršavanja algoritma u sekundama, osim ako je u tekstu drukčije naznačeno.

Poglavlje 2

Početna razmatranja

U ovom poglavlju ćemo proučiti problem u jednoj dimenziji (koji je gotovo trivijalan) i "brute-force" algoritam za dvije dimenzije.

2.1 Jednodimenzionalni slučaj

U slučaju $S \subset \mathbb{R}$, problem se svodi na pronalaženje dvaju realnih brojeva iz S čija razlika ima najmanju apsolutnu vrijednost. To očito možemo riješiti računajući razliku svaka dva broja iz S, što zahtijeva računanje $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ razlikā.[1]

Međutim, ako prethodno uzlazno sortiramo niz brojeva, problem se značajno pojednostavljuje – dovoljno je samo provjeriti razlike svaka dva "susjedna" broja. Sljedeći kôd je implementacija algoritma koji smo upravo opisali.

```
1  pair<double, double> find_closest(vector<double> S) {
2      sort(S.begin(), S.end());
3      pair<double, double> closest = make_pair(S[0], S[1]);
4      double best = S[1] - S[0];
5      for(size_t i = 1; i < S.size()-1; ++i) {
6          if(S[i+1] - S[i] < best) {
7              best = S[i+1] - S[i];
8              closest = make_pair(S[i], S[i+1]);
9          }
10     }
11     return closest;
12 }
```

[1] Veličinu ulaznih podataka označavamo s n.

Možemo jednostavno odrediti složenost ovog algoritma:

1. Čitav niz se kopira (kako funkcija ne bi mijenjala originalni niz) – $O(n)$.
2. Sortiramo kopirani niz – $O(n \log n)$.
3. Jedan prolazak kroz čitav niz za određivanje najmanje razlike – $O(n)$.

Dakle, složenost čitavog algoritma je $O(n \log n)$.

2.2 Dvodimenzionalni slučaj

Sada imamo $S \subset \mathbb{R}^2$. Za razliku od jednodimenzionalnog slučaja, nemamo totalni uređaj po kojem bismo sortirali točke u S. Osim toga, udaljenost između dviju točaka nije samo razlika koordinatā već se računa po formuli

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

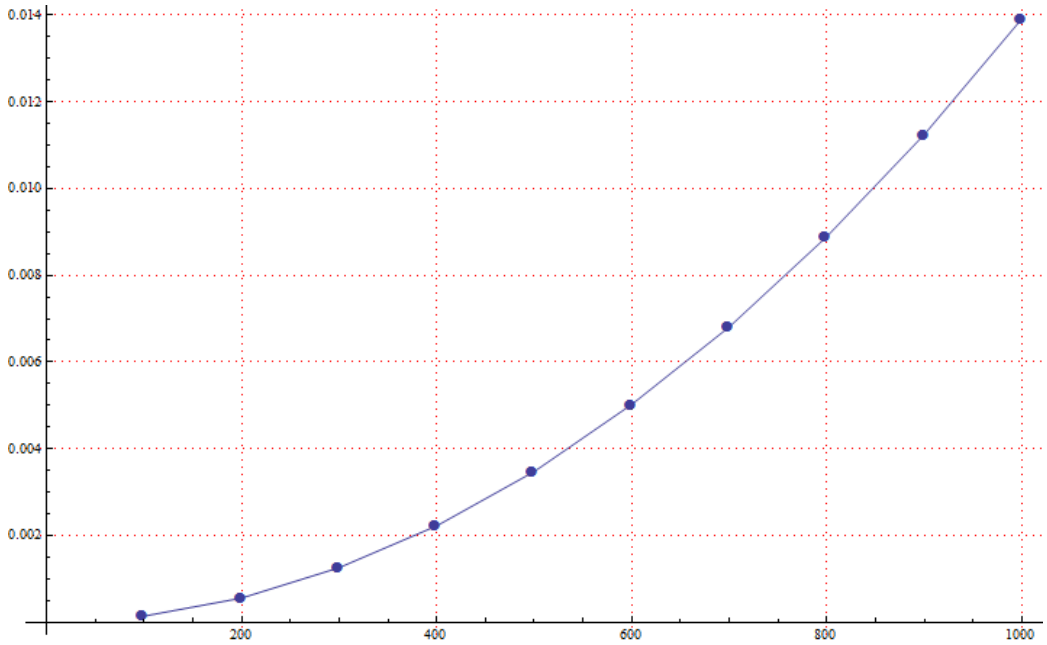
Naravno, za uspoređivanje je dovoljno izračunati samo $(x_1 - x_2)^2 + (y_1 - y_2)^2$ zato što korijen čuva uređaj, no moramo paziti da u slučaju kad nam zbilja treba udaljenost izračunamo drugi korijen od dobivene vrijednosti.

Kao i u jednoj dimenziji, najočitiji način da pronađemo najbliži par je da izračunamo svih $\binom{n}{2}$ udaljenosti među točkama i uzmemo minimum. Slijedi implementacija tog algoritma.[2]

```
result brute_force(const vector<point>& S,
                   const size_t& l, const size_t& r) {
    result closest(S[l], S[l+1]);
    for(size_t i = l; i < r-1; ++i) {
        for(size_t j = i+1; j < r; ++j) {
            double temp(distance(S[i], S[j]));
            if(temp < closest.d) {
                closest.d = temp;
                closest.a = S[i];
                closest.b = S[j];
            }
        }
    }
    return closest;
}
```

[2] Varijable `l` i `r` su trenutno nepotrebne, ali ćemo ih koristiti kasnije, kad ovu funkciju budemo pozivali u drugim algoritmima. Također, u svim algoritmima ćemo kao ulaz primati STL `vector` podataka tipa `point`.

Kao što smo već rekli, ovaj algoritam provjerava svih $\binom{n}{2}$ udaljenosti među točkama, prema tome složenost mu je $O(n^2)$. To je očito i iz sljedećeg grafa koji prikazuje mjerenje trajanja rada ovog algoritma na ulaznim podacima duljinā od 100 do 1000 točaka.



Ovaj algoritam je prespor za skupove veće od nekoliko tisuća točaka. U sljedećem poglavlju ćemo vidjeti kako i u dvije dimenzije možemo iskoristiti sortiranje da ostvarimo algoritam manje složenosti, ali prije toga ćemo iskazati teorem koji ćemo koristiti za računanje složenosti naknadnih algoritama. Dokaz teorema se može pronaći u [1].

Teorem 2.2.1 (Master teorem)**.** *Ako je T rekurzivna relacija za koju vrijedi*

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),\ a \geq 1, b > 1,$$

tada vrijedi:

1. *Ako je $f(n) = O(n^c)$, $c < \log_b a$, tada je $T(n) = \Theta(n^{\log_b a})$.*
2. *Ako $\exists\, k \geq 0$ t.d. $f(n) = \Theta(n^{\log_b a} \log^k n)$, tada je $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *Ako je $f(n) = \Omega(n^c)$, $c > \log_b a$, te $\exists\, k < 1, n_0 \in \mathbb{N}$ t.d. $\forall n > n_0$, $af\left(\frac{n}{b}\right) \leq kf(n)$, tada je $T(n) = \Theta(f(n))$.*

Poglavlje 3

Divide-and-conquer

Kao što smo najavili u prethodnom poglavlju, ponovno možemo iskoristiti sortiranje za izvedbu efikasnijeg algoritma. Treba napomenuti da to nije očito na prvi pogled – sortiranje po x- ili y-osi samo po sebi nam ne pomaže, pošto te dvije vrijednosti mogu biti potpuno nezavisne, čime nam bliskost u x-osi ne ukazuje ni na kakvu bliskost u y-osi ili obrnuto, a time ni bliskost po euklidskoj metrici.

3.1 Prvi pristup

Ako ulazni vektor S sadrži manje od četiri točke, najbliži par pronađemo algoritmom `brute_force`. Inače, S najprije sortiramo po x koordinati. Potom, vektor dijelimo na lijevu i desnu polovicu, označimo ih sa S_l i S_r, te nad obje rekurzivno pozivamo algoritam. Pretpostavljajući da nakon toga znamo najbliži par točaka u S_l i najbliži par u S_r, obilježimo onaj koji je od ta dva bliži kao najbliži dosad pronađeni i označimo udaljenost tih dviju točaka s δ. Naravno, time nismo gotovi – postoji mogućnost da je "globalni" najbliži par neki kojemu je jedna točka u S_l, a druga u S_r.

Označimo s x_m srednju vrijednost x koordinatā zadnje točke u S_l i prve u S_r. Uočimo da, ako postoji par točaka čija međusobna udaljenost je manja od δ, vrijednost x koordinate jedne od te dvije točke se mora nalaziti u intervalu $[x_m - \delta, x_m]$, a druge u intervalu $[x_m, x_m + \delta]$. Prema tome, da bismo provjerili postojanje takvog para točaka, dovoljno je uzeti u obzir samo točke čija vrijednost x koordinate se nalazi unutar $[x_m - \delta, x_m + \delta]$. Označimo vektor koji sadrži samo te točke sa S_m.

Nažalost, ovdje nailazimo na početni problem – izgleda da moramo provjeriti međusobnu udaljenost svake dvije točke u S_m. Pošto u najgorem slučaju može biti $S_m = S$, time bi složenost ovog algoritma ponovno bila $O(n^2)$. Međutim, sljedeća lema ukazuje na jednu neočekivanu posljedicu onoga što smo u algoritmu dosad obavili.

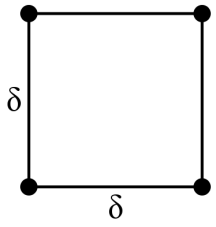
Lema 3.1.1. *Uz gore danu definiciju δ i S_m, ako postoje točke $a, b \in S_m$ t.d. $d(a,b) < \delta$, tada je u S_m točka a unutar sedam najbližih točaka točki b po vrijednosti y koordinate.*

Drugim riječima, ako sortiramo vektor S_m po y koordinati, za svaku točku u S_m je dovoljno provjeriti udaljenost od samo sedam točaka koje ju neposredno slijede.

Dokaz. Pretpostavimo da postoje točke $a = (x_a, y_a)$ i $b = (x_b, y_b)$ iz S_m koje zadovoljavaju $d(a,b) < \delta$. Tada specijalno vrijedi i $|y_a - y_b| < \delta$. Pošto također vrijedi $x_a, x_b \in [x_m - \delta, x_m + \delta]$, to znači da postoji pravokutnik dimenzijā $2\delta \times \delta$ centriran po x-osi oko x_m koji sadrži točke a i b.

B.s.o. pretpostavimo da je $x_a \in [x_m - \delta, x_m]$ te $x_b \in [x_m, x_m + \delta]$. Točka a se nalazi unutar kvadrata dimenzijā $\delta \times \delta$ koji je lijeva polovica pravokutnika koji sadrži a i b, a točka b se nalazi u $\delta \times \delta$ kvadratu koji je njegova desna polovica. Također, lijevi kvadrat može, osim a, sadržavati samo točke iz S_l, dok desni, osim b, može sadržavati samo točke iz S_r.

Znamo da u S_l ne postoji par točaka koje su međusobno udaljene manje od δ, a isto vrijedi i za S_r. Najveći broj točaka koje se mogu nalaziti unutar kvadrata dimenzijā $\delta \times \delta$, a da je udaljenost između svake dvije veća ili jednaka od δ, je 4, kao što je prikazano na sljedećoj slici:



Dakle, najveći mogući broj točaka u lijevom kvadratu je 4, a isto vrijedi i za desni kvadrat. Prema tome, najveći mogući broj točaka unutar pravokutnika koji sadrži a i b je 8. To znači da, ako sortiramo točke u S_m po vrijednosti y koordinate, u najgorem slučaju će se točka a nalaziti sedam mjestā ispred ili iza točke b. □

Sve što je ostalo za obaviti je implementirati opisani algoritam. Zbog jednostavnosti ćemo pretpostaviti da se algoritmu predaje već sortirani vektor točaka.[1] Funkcija `find_left` koristi varijantu binarnog pretraživanja da bi pronašla indeks najljevije točke čija x koordinata je veća ili jednaka $x_m - \delta$, a funkcija `find_right` slično pronalazi indeks najljevije točke čija x koordinata je veća od $x_m + \delta$.[2]

[1] U mjerenje vremena je, naravno, uključeno i sortiranje vektora prije poziva algoritma.

[2] Zbog toga što copy konstruktor vektora kopira od prvog indeksa uključivo do drugog indeksa isključivo.

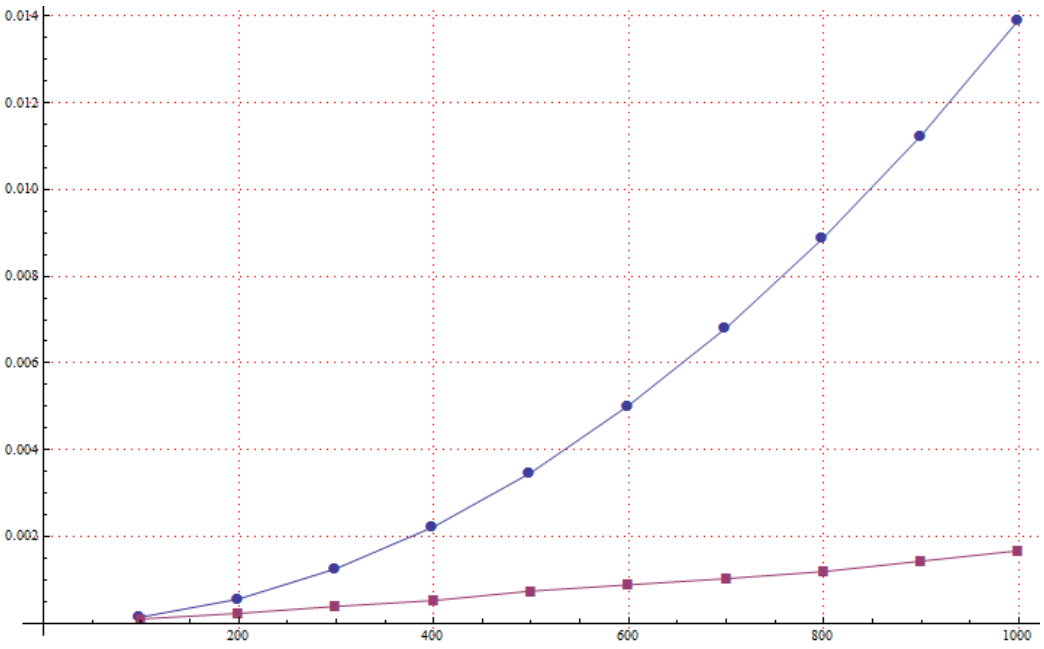
```
result Divide_and_Conquer(const vector<point>& S,
                          const size_t& l, const size_t& r) {
    if(r - l < 4) return brute_force(S, l, r);
    size_t mid = (l + r) / 2;
    double x_m = (S[mid-1].first + S[mid].first)/2.0;
    result closest_L(Divide_and_Conquer(S, l, mid));
    result closest_R(Divide_and_Conquer(S, mid, r));
    result closest(closest_L.d < closest_R.d ? closest_L : closest_R);
    double delta = sqrt(closest.d);
    size_t left(find_left(S, l, mid, x_m - delta)),
        right(find_right(S, r, mid, x_m + delta));
    vector<point> S_m(S.begin() + left, S.begin() + right);
    if(S_m.size() > 1) {
        sort(S_m.begin(), S_m.end(), compare_y);
        for(size_t i = 0; i < S_m.size() - 1; ++i) {
            for(size_t j = i + 1; (j < i+8) && (j < S_m.size()); ++j) {
                double temp = distance(S_m[i], S_m[j]);
                if(temp < closest.d) {
                    closest.a = S_m[i];
                    closest.b = S_m[j];
                    closest.d = temp;
                }
            }
        }
    }
    return closest;
}
```

Prethodno smo napomenuli da je u najgorem slučaju $S_m = S$. To znači da je sortiranje S_m koje obavljamo u rekurziji $O(n \log n)$. Prema tome, vrijeme izvršavanja algoritma možemo prikazati rekurzijom

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n).$$

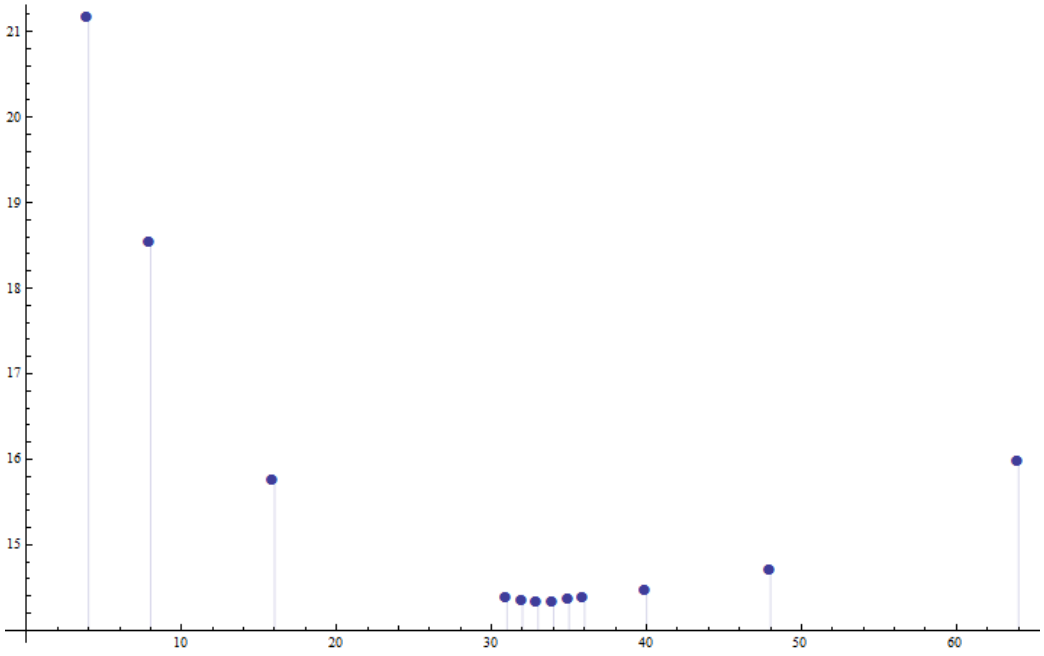
Po slučaju 2. iz Master teorema imamo $a = b = 2$, $\log_b a = 1$, $k = 1$, što znači da je $T(n) = O(n \log^2 n)$. Sortiranje čitavog vektora prije poziva algoritma je $O(n \log n)$, pa je ukupna složenost $O(n \log^2 n)$.

Sada ćemo usporediti mjerenja ovog algoritma s algoritmom `brute_force`.

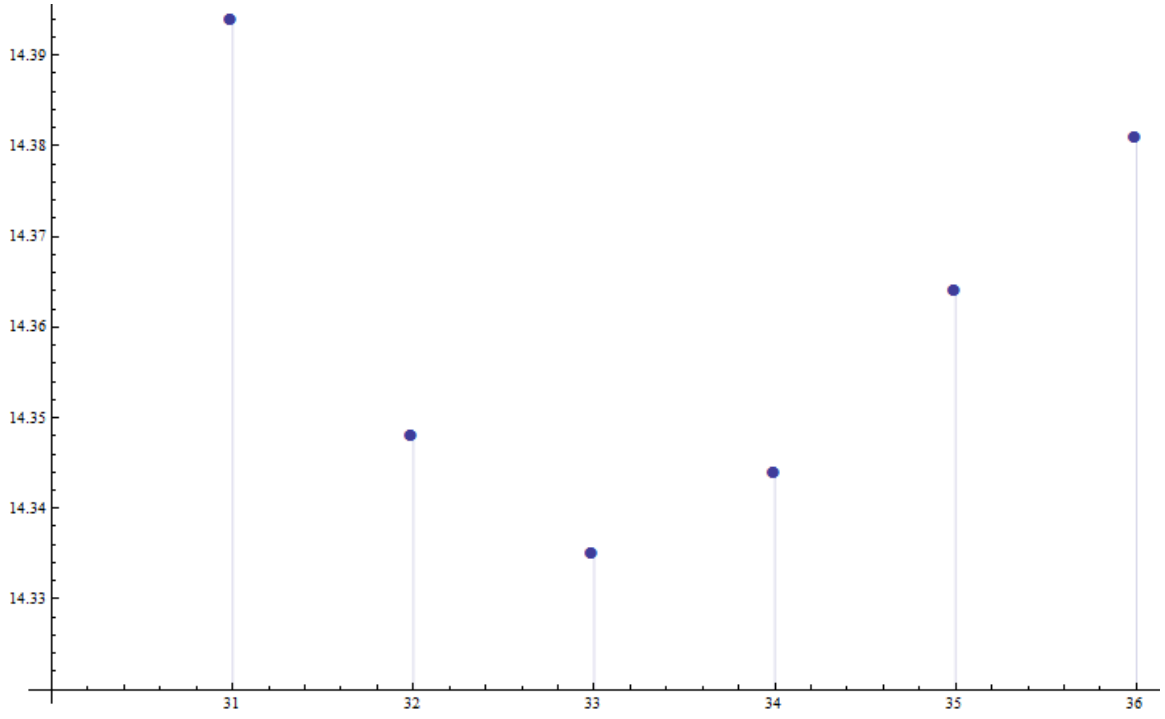


Slika 3.1: ● `brute_force`; ■ `Divide_and_Conquer`

Kao što možemo vidjeti, razlika u brzini je drastična. Svejedno, možemo pokušati dodatno ubrzati algoritam povećavajući maksimalnu duljinu na kojoj pozivamo `brute_force`.



Prethodni graf prikazuje vrijeme potrebno algoritmu da završi na vektorima duljinā od 128 do 4096 za različite vrijednosti cjelobrojnog parametra na liniji 4.[3] Prikažimo samo vrijednosti od 31 do 36:

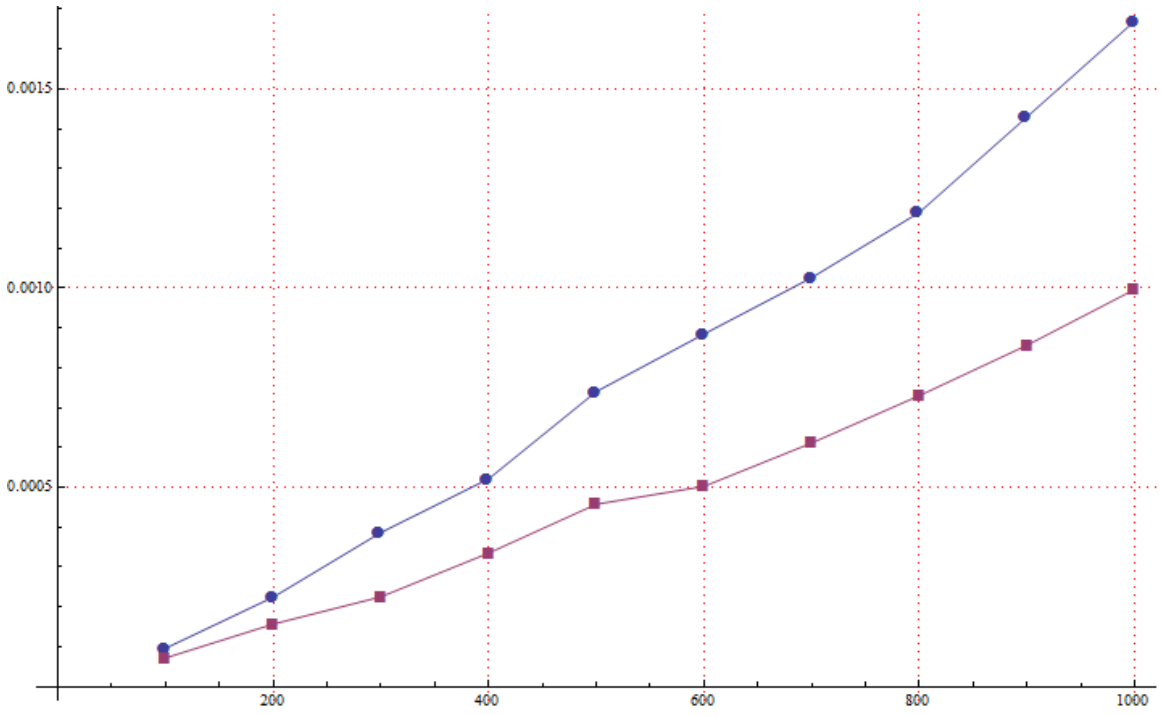


Iz mjerenja je očito da se najveća brzina postiže ako `brute_force` pozivamo kada je duljina ispod 33. Stoga, liniju 4 u kodu ćemo zamijeniti sa

```
4     if(r - l < 33) return brute_force(S, l, r);
```

Idući graf uspoređuje mjerenja algoritma za granicu 4 i za granicu 33.

[3] U petlji `for(size_t i = 128; i <= 4096; ++i)` su generirani vektori duljinā `i` te je algoritam izvršavan na njima. Vrijeme potrebno za generiranje brojeva nije oduzimano jer je konstantno.



Slika 3.2: ● granica 4; ■ granica 33

3.2 Poboljšanje složenosti

Složenost prethodnog algoritma je $O(n \log^2 n)$ zbog toga što obavljamo sortiranje vektora duljine $O(n)$ unutar rekurzivnog poziva. Međutim, sortiranje po y koordinati možemo također obaviti prije pozivanja algoritma.

Dakle, prije izvršavanja algoritma ćemo kopirati ulazni vektor S – označimo kopiju sa S'. Potom S sortiramo po x koordinati, a S' po y koordinati. Algoritmu kao ulaz predajemo oba vektora. Pri rekurzivnom pozivanju algoritma na S_l i S_r, potrebno je iz S' konstruirati vektore S'_l i S'_r koji sadrže točke iz S_l odnosno S_r, sortirane po y koordinati. To možemo ostvariti postupkom "obrnutim" od spajanja dvije sortirane liste u jednu. Inicijalizirajmo S'_l i S'_r kao prazne vektore. Iterirajući od početka do kraja S', ako je x koordinata točke manja od x_m dodajemo ju na kraj S'_l, inače ju dodajemo na kraj S'_r. Pošto su točke u S' sortirane po y koordinati te smo iterirali redom od početka do kraja S', točke u S'_l i S'_r će ostati sortirane po y koordinati.

Isti postupak možemo iskoristiti da konstruiramo S_m sortiran po y koordinati – inicijaliziramo S_m kao prazni vektor, potom iteriramo po S' i točku dodajemo u S_m ako joj je x koordinata unutar $[x_m - \delta, x_m + \delta]$. Slijedi implementacija ovog algoritma. Kao i u pret-

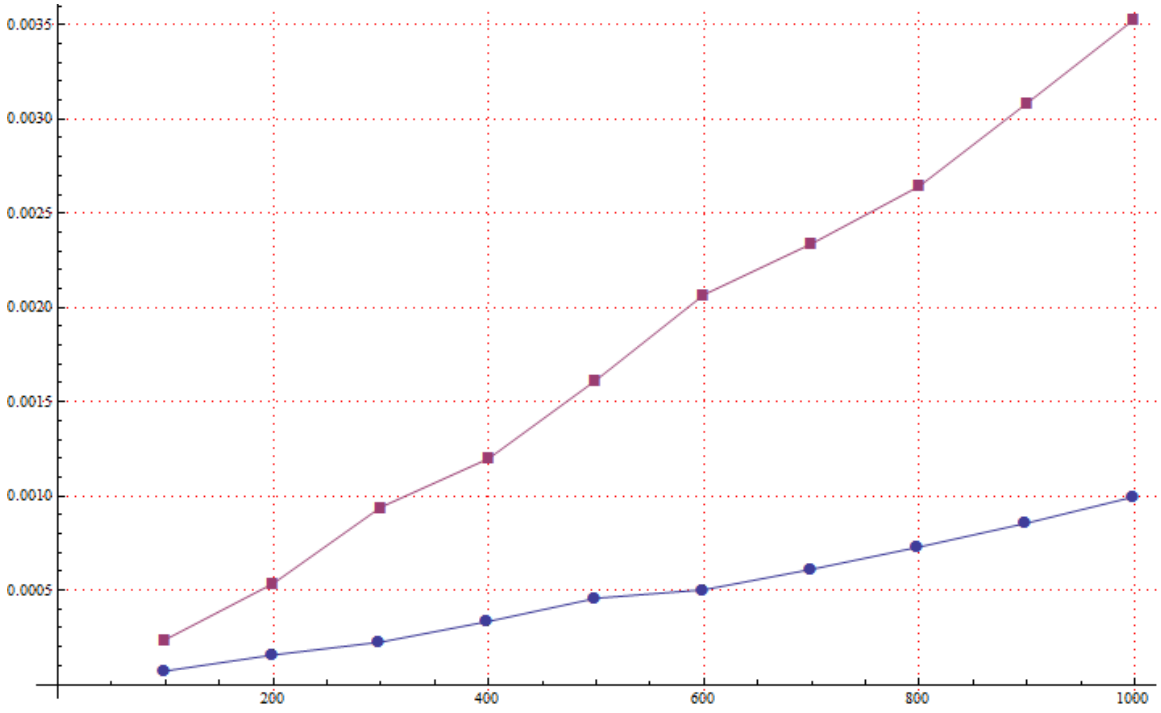
hodnom slučaju, pretpostavljamo da su S i S' sortirani prije pozivanja algoritma (u kodu je "S'" zamijenjeno sa "S2").

```
result D_and_C_2(const vector<point>& S, const vector<point>& S2,
                 const size_t& l, const size_t& r) {
    if(r - l < 4) return brute_force(S, l, r);
    size_t mid = (l + r) / 2;
    double x_m = (S[mid-1].first + S[mid].first)/2.0;
    vector<point> S2_l, S2_r;
    for(auto& p : S2) {
        if(p.first < x_m) S2_l.push_back(p);
        else S2_r.push_back(p);
    }
    result closest_L(D_and_C_2(S, S2_l, l, mid));
    result closest_R(D_and_C_2(S, S2_r, mid, r));
    result closest(closest_L.d < closest_R.d ? closest_L : closest_R);
    double delta = sqrt(closest.d),
        llimit = x_m - delta,
        rlimit = x_m + delta;
    vector<point> S_m;
    for(auto& p : S2)
        if(p.first >= llimit && p.first <= rlimit) S_m.push_back(p);
    if(S_m.size() > 1) {
        for(size_t i = 0; i < S_m.size()-1; ++i) {
            for(size_t j = i+1; (j < i+8) && (j < S_m.size()); ++j) {
                double tempdist = distance(S_m[i], S_m[j]);
                if(tempdist < closest.d) {
                    closest.a = S_m[i];
                    closest.b = S_m[j];
                    closest.d = tempdist;
                }
            }
        }
    }
    return closest;
}
```

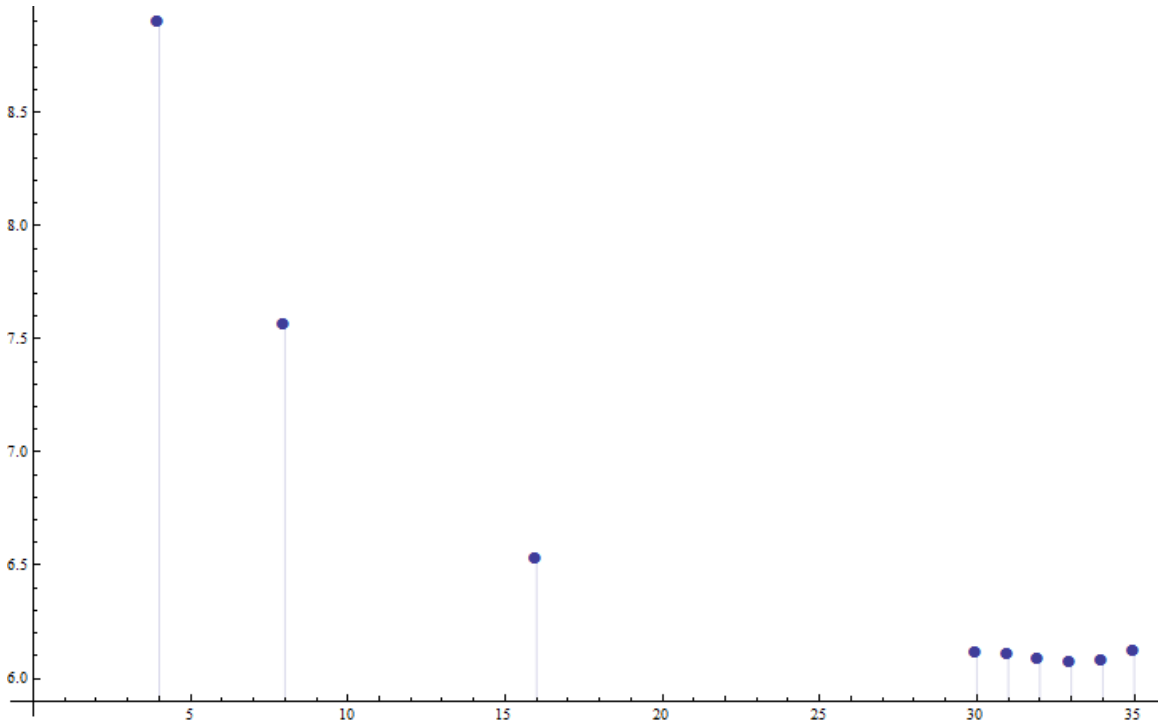
Sve operacije osim rekurzivnih poziva u ovom algoritmu su $O(n)$. Vrijeme izvršavanja možemo prikazati rekurzijom

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

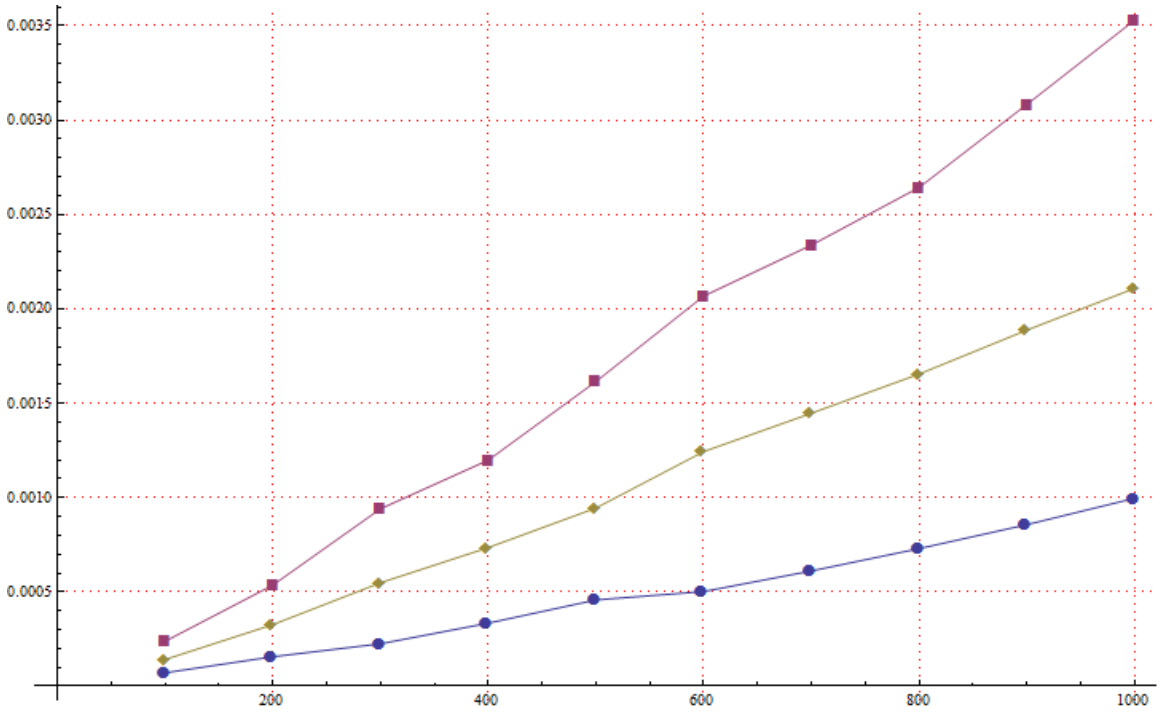
Po slučaju 2. iz Master teorema imamo $a = b = 2$, $\log_b a = 1$, $k = 0$, što znači da je $T(n) = O(n \log n)$. Sortiranje S i S' prije poziva algoritma je $O(n \log n)$. Prema tome, ukupna složenost je $O(n \log n)$. Sljedeći graf uspoređuje ovaj algoritam s najboljom verzijom prethodnog algoritma.



Slika 3.3: ● `Divide_and_Conquer`; ■ `D_and_C_2`



Drugi graf prikazuje rezultate pokušaja ubrzanja algoritma istim postupkom kao i prije – mijenjanjem granice za pozivanje `brute_force` algoritma. Za ovaj test smo `D_and_C_2` pozivali na vektorima duljina od 128 do 2048 (zbog sporijeg izvršavanja u odnosu na `Divide_and_Conquer`). Iz grafa se vidi da je, kao i za `Divide_and_Conquer`, najveća brzina izmjerena za granicu 33. Sljedeći graf prikazuje rezultate mjerenja modificiranog algoritma.



Slika 3.4: ● `Divide_and_Conquer`; ■ `D_and_C_2` (4); ◆ `D_and_C_2` (33)

Možemo primijetiti da, usprkos smanjenoj složenosti, algoritam `D_and_C_2` je sporiji od algoritma `Divide_and_Conquer`. Dodatno testiranje je pokazalo da je čak i za vektore duljine 10,000,000 `Divide_and_Conquer` značajno brži od `D_and_C_2`. Ako detaljno proanaliziramo ova dva algoritma, očito je zašto bi situacija bila takva.

- Ocjena $O(n \log^2 n)$ za `Divide_and_Conquer` je točna uz pretpostavku da je duljina vektora S_m $O(n)$. Međutim, osim ako su ulazni podaci posebno konstruirani da dovedu do takve situacije, duljina S_m će u praksi s duljinom ulaznih podataka rasti mnogo sporije nego $O(n)$. Time će i operacija sortiranja S_m po y koordinati biti brža od $O(n \log n)$.

- U `Divide_and_Conquer` pri konstruiranju vektora S_m copy konstruktor iterira samo po onim točkama u S koje će biti dodane u S_m. Ta operacija je i dalje $O(n)$ u najgorem slučaju, ali, uz prethodnu opasku, stvarni broj točaka po kojima treba iterirati će u praksi biti puno manji. Za razliku od toga, bez obzira na broj točaka koje trebaju biti dodane u S_m, u `D_and_C_2` se iterira po čitavom vektoru S' i za svaku točku posebno provjerava uvjete na x koordinatu za dodavanje u S_m. Ova operacija, za razliku od analogne u `Divide_and_Conquer`, nije samo $O(n)$ u najgorem slučaju, već je $\Theta(n)$ bez obzira na broj točaka koje zbilja trebaju biti dodane u S_m.

Za kraj ovog poglavlja, spomenimo jedan detalj o algoritmu `D_and_C_2` koji smo prešutno zanemarili. Promotrimo situaciju kada u S postoji više točaka čija vrijednost x koordinate je x_m. Tada će neke od tih točaka biti u S_l a neke u S_r (koje točno, ovisi o implementaciji algoritma kojim smo sortirali S te o tome koliko ih je nakon toga lijevo od sredine vektora a koliko desno), ali će sve biti u S'_r. Međutim, to ne narušava niti korektnost algoritma, niti složenost $O(n \log n)$.

Što se korektnosti tiče, razmotrimo granu rekurzije u kojoj se zbilja dogodi da S'_l sadrži manje točaka nego S_r, a S'_r više nego S_r; nazovimo ju "A". Pretpostavimo da se u svim daljnjim granama rekurzije ovo ne dogodi.[4] Kako bismo razlikovali varijable iz različitih granā rekurzije, od sad ćemo ih obilježavati sa "[simbol rekurzije]ime varijable". Promotrimo grane rekurzije koje pozivamo nad $([A]S_l, [A]S'_l)$ i $([A]S_r, [A]S'_r)$; nazovimo ih "B" i "C". Udaljenost najbližeg para kojeg pronađe grana B će sigurno biti manja ili jednaka udaljenosti najbližeg para svih točaka koje su lijevo od $[A]x_m$. Grana C će pronaći najbliži par točaka u $[A]S_r$, osim ako vektor $[C]S_m$ obuhvati točke čija x koordinata je $[A]x_m$ te postoji par točaka u $[A]S'_r$ koji je bliži od najbližeg para u $[A]S_r$ (čija barem jedna točka tada mora biti točka iz $[A]S_l$ čija x koordinata je $[A]x_m$) te su te dvije točke u $[C]S_m$ manje od osam mjestā jedna od druge. Prema tome, udaljenost najbližeg para kojeg pronađe grana C će sigurno biti manja ili jednaka udaljenosti najbližeg para u $[A]S_r$, pa specijalno i svih točaka koje su desno od $[A]x_m$. Sada se vraćamo u granu A (od sad su sve varijable u toj grani, pa nećemo više pisati prefiks), uzimamo manju od ovih dvaju udaljenosti i označavamo ju s δ. Konstruiramo vektor S_m koji sadrži točke čija x koordinata je unutar $[x_m - \delta, x_m + \delta]$, sortirane po y koordinati. Ako postoji par točaka $a, b \in S_m$ koje su međusobno bliže od δ, vrijedi točno jedno od sljedećeg:

1. x koordinate obaju točaka su x_m.
2. x koordinata jedne točke je jednaka, a druge manja od x_m.
3. x koordinata jedne točke je jednaka, a druge veća od x_m.

[4] Primijetimo da će rekurzivni pozivi sigurno završiti pozivom algoritma `brute_force` u kojemu se ovo ne može dogoditi te da, ako dokažemo korektnost najdublje grane na kojoj se ova situacija dogodi, tada korektnost svih grana "iznad" nje u kojima se eventualno dogodi ista situacija slijedi po indukciji.

4. x koordinata jedne točke je manja, a druge veća od x_m.

Prisjetimo se najprije dokaza leme 3.1.1 – u kvadratu dimenzijā $\delta \times \delta$ mogu biti najviše četiri točke čija međusobna udaljenost je veća ili jednaka od δ, i to ako se nalaze točno na vrhovima tog kvadrata. Međutim, dva vrha $\delta \times \delta$ kvadrata koji je lijeva polovica $2\delta \times \delta$ pravokutnika koji sadrži točke a i b se nalaze na vertikali $x = x_m$, kao i dva vrha $\delta \times \delta$ kvadrata koji je njegova desna polovica. Znamo da lijevo, kao ni desno, od x_m ne postoji par točaka bliži od δ. Prema tome, najveći broj točaka lijevo od x_m koje se mogu nalaziti u lijevoj polovici $2\delta \times \delta$ pravokutnika je 3, kao i najveći broj točaka desno od x_m koje se mogu nalaziti u desnoj polovici. Dakle, najveći mogući broj točaka čija x koordinata je različita od x_m unutar $2\delta \times \delta$ pravokutnika je 6.

Pretpostavimo sada da vrijedi prvi gore navedeni slučaj. B.s.o. pretpostavimo da je y koordinata točke a manja od y koordinate točke b. Čak i ako su točke a i b na dnu, odnosno vrhu $2\delta \times \delta$ pravokutnika, u vektoru S_m je između njih najviše šest točaka čija x koordinata je različita od x_m. Prema tome, ako ne postoji točka između a i b čija x koordinata je x_m, tada su a i b najviše sedam mjestā jedna od druge. Ako postoji neka takva točka c, tada je udaljenost između a i c manja nego između a i b, pa nas par (a, b) niti ne zanima, a broj točaka čija x koordinata je različita od x_m koje su između a i c je sigurno manji ili jednak broju takvih točaka između a i b.

Pretpostavimo da vrijedi drugi slučaj. B.s.o. pretpostavimo da je b točka čija x koordinata je manja od x_m. Znamo da osim b u $2\delta \times \delta$ pravokutniku koji sadrži a i b postoji najviše pet drugih točaka čija x koordinata je različita od x_m. Ako ne postoji točka između a i b čija x koordinata je x_m, tada su a i b najviše šest mjestā jedna od druge. Ako postoji neka takva točka c, udaljenost između a i c je manja nego između a i b. Potpuno analogan argument možemo iskoristiti za treći slučaj.

Ako vrijedi četvrti slučaj, tada između a i b mogu biti najviše četiri točke čija x koordinata je različita od x_m. Ako postoji točka c čija x koordinata je x_m koja je između a i b, tada je i udaljenost između a i c i udaljenost između b i c manja nego udaljenost između a i b.

Prema svemu izrečenom, zaključak je da je i dalje dovoljno provjeriti samo sedam sljedbenika svake točke u S_m, dakle algoritam je korektan.

Da pokažemo da je složenost algoritma i dalje $O(n \log n)$, dovoljno je promotriti najgori, "patološki" slučaj kada je vrijednost x koordinate svih točaka u S ista. Nazovimo pozive rekurzije nad (S_l, S'_l) "lijeva grana", a pozive rekurzije nad (S_r, S'_r) "desna grana". Uočimo da, ako je duljina vektora S' jednaka 0 u nekoj grani rekurzije, tada je duljina S' također 0 i za lijevu i za desnu granu te grane. U nultom nivou rekurzije (početni poziv algoritma), duljina vektora S' je n. Za lijevu granu i sve njene daljnje grane, duljina S' će biti 0. Za desnu granu će duljina S' biti n. Za lijevu granu desne grane je duljina S' ponovno 0, a za desnu n. Ovo se rekurzivno nastavlja do najdubljeg nivoa rekurzije – duljina S' je n samo za "desne grane desnih granā". Iz toga zaključujemo da na svakom nivou

rekurzije postoji točno jedan poziv ("najdesnija grana") za koji je duljina S' jednaka n, a za sve ostale je 0. Dakle, postoji točno onoliko pozivā rekurzije u kojima je duljina S' jednaka n koliko ima nivoā rekurzije. U svakom takvom pozivu će se iterirati po vektoru duljine n dva puta – jednom za konstruiranje S'_l i S'_r, jednom za konstruiranje S_m. Pošto niz S dijelimo na pola pri svakom pozivu rekurzije, broj nivoā rekurzije je $\lceil \log_2 n \rceil = O(\log n)$. Prema tome, iteriranje po vektoru duljine n će se obaviti $2 * O(\log n) = O(\log n)$ puta, što je $O(n \log n)$.

Poglavlje 4

Slučajno uzorkovanje

U ovom poglavlju proučavamo algoritme koji koriste slučajno uzorkovanje točaka u S za dobivanje približne procjene udaljenosti najbližeg para te potom egzaktno pronalaze najbliži globalni par koristeći dobivenu procjenu za smanjenje broja parova točaka čija međusobna udaljenost se treba provjeriti. Vidjet ćemo da ovim pristupom algoritmi mogu postići očekivanu složenost $O(n)$.

4.1 Rabinov algoritam

U [5], Michael Oser Rabin predstavlja algoritam koji je veoma bitan u povijesti računarstva. Naime, to je bio prvi nedeterministički algoritam (algoritam koji se oslanja na generiranje pseudo-slučajnih brojeva) koji ima manju očekivanu složenost od svih poznatih determinističkih algoritama. Sada ćemo opisati spomenuti algoritam.

Iz ulaznog vektora točaka S duljine n odaberimo slučajni uzorak od $\lceil \sqrt{n} \rceil$ različitih točaka.[1] Algoritmom `brute_force` odredimo najbliži par točaka u ovom skupu i označimo njihovu međusobnu udaljenost s δ. Ako je $\delta = 0$, vratimo pronađeni par. Inače, podijelimo čitavu ravninu $\mathbb{R}^2$ na $\delta \times \delta$ kvadrate. Preciznije rečeno, definirajmo skup

$$G := \Big\{ [i\delta, (i+1)\delta\rangle \times [j\delta, (j+1)\delta\rangle \;\Big|\; i, j \in \mathbb{Z} \Big\}.$$

Ovako definirani skup G ćemo zvati *δ-mreža*.

Za svaku točku u S odredimo u kojem kvadratu u δ-mreži se nalazi. Zatim, za svaki kvadrat u δ-mreži u kojem se nalazi barem jedna točka iz S odredimo pomoću algoritma `brute_force` najbliži par u skupu svih točaka koje se nalaze u tom ili jednom od osam susjednih kvadratā.[2] Najbliži od svih parova točaka koje pronađemo na ovaj način je najbliži par točaka u skupu S.

[1] Pod tim mislimo na različite indekse u S. Ako S sadrži duplikate, isti se mogu pojaviti i u uzorku.

[2] Dva kvadrata su "susjedna" ako imaju zajedničku stranicu ili zajednički vrh.

Analiza složenosti Rabinovog algoritma je kompleksna. Očito je da je složenost prvog koraka algoritma $O(n)$ – pozivamo algoritam složenosti $O(n^2)$ na vektoru duljine $O(\sqrt{n})$. Pretpostavimo zasad da i drugi korak, određivanje pripadajućeg kvadrata u δ-mreži za svaku točku, možemo obaviti u $O(n)$ te da imamo mogućnost "iteriranja" samo po onim kvadratima u kojima se nalazi barem jedna točka iz S. Treći korak je problematičan – trebalo bi procijeniti očekivani broj točaka u kvadratu dimenzijā $3\delta \times 3\delta$. Ovdje se nećemo baviti ovom analizom nego ćemo prikazati način implementacije i testiranje algoritma.

Da bismo mogli implementirati algoritam, najprije moramo nekim tipom podataka predstaviti δ-mrežu. Kvadrate u δ-mreži ćemo predstaviti parovima cijelih brojeva s predznakom (praktički, brojevi i, j iz definicije skupa G):

```
#define gridindex pair<int64_t, int64_t>
```

Za danu točku $p = (p_x, p_y)$ u konstantnom vremenu možemo odrediti u kojem kvadratu se nalazi računajući $\left(\left\lfloor \frac{p_x}{\delta} \right\rfloor, \left\lfloor \frac{p_y}{\delta} \right\rfloor\right)$. Dakle:

```
inline gridindex gridbin(const point& p, const double& d) {
    return make_pair(floor(p.first/d), floor(p.second/d));
}
```

Sada očito možemo odrediti pripadajući kvadrat za svaku točku iz S u $O(n)$. Međutim, još nemamo mogućnost iteriranja samo po onim kvadratima koji sadrže barem jednu točku iz S. No, rješenje je jednostavno – podatke tipa `gridindex` možemo koristiti kao ključ u mapi te im pridružiti vektor točaka koje se nalaze u njima. Najprije ćemo proći kroz vektor S i popuniti mapu odgovarajućim podacima, zatim još jednom prolazimo kroz S i za svaku točku $p = (p_x, p_y)$ dobavljamo vektor svih točaka koje se nalaze u kvadratu $\left(\left\lfloor \frac{p_x}{\delta} \right\rfloor, \left\lfloor \frac{p_y}{\delta} \right\rfloor\right)$ ili susjednim kvadratima, koje možemo lako dobiti oduzimajući/dodajući 1 od/na $\left\lfloor \frac{p_x}{\delta} \right\rfloor$ i/ili $\left\lfloor \frac{p_y}{\delta} \right\rfloor$ te dobavljajući odgovarajući vektor za taj ključ iz mape.

Kako bi indeksiranje bilo konstantna a ne logaritamska operacija, umjesto `map` ćemo koristiti `unordered_map`. Za to najprije moramo implementirati `hash` operaciju za podatke tipa `gridindex` (pošto ista u STL-u nije implementirana za `pair`). U tu svrhu koristimo sljedeću funkciju (naknadno ćemo objasniti izbor funkcije):

```
namespace std {
    template <>
    struct hash<gridindex> {
        size_t operator() (const gridindex& k) const {
            size_t n = (k.first >= 0 ? 2*k.first : -2*k.first - 1),
                m = (k.second >= 0 ? 2*k.second : -2*k.second - 1);
            return (n >= m ? n*n + n + m : n + m*m);
        }
    };
}
```

Također, u svrhu dodatne optimizacije ćemo, osim vektora točaka, ključevima pridruživati i jednu `bool` varijablu koja će obilježavati je li taj kvadrat već provjeren.

```
15 class grid {
16     unordered_map<gridindex, pair<vector<point>, bool>> data;
17     double d;
18 
19 public:
20     grid(const vector<point>& S, const double& _d) : d(_d) {
21         for(auto& p : S)
22             data[gridbin(p, d)].first.push_back(p);
23         for(auto& p : data) p.second.second = true;
24     }
25 
26     bool& operator[] (const point& p) {
27         return (data.at(gridbin(p, d))).second;
28     }
29 
30     vector<point> operator() (const point& p) {
31         gridindex p_index(gridbin(p, d));
32         vector<point> ret((data.at(p_index)).first);
33         vector<gridindex> v(8);
34         v[0] = make_pair(p_index.first, p_index.second + 1);
35         v[1] = make_pair(p_index.first + 1, p_index.second);
36         v[2] = make_pair(p_index.first, p_index.second - 1);
37         v[3] = make_pair(p_index.first - 1, p_index.second);
38         v[4] = make_pair(v[1].first, v[0].second);
39         v[5] = make_pair(v[1].first, v[2].second);
40         v[6] = make_pair(v[3].first, v[0].second);
41         v[7] = make_pair(v[3].first, v[2].second);
42         for(auto& k : v) {
43             auto it = data.find(k);
44             if(it != data.end() && it->second.second) {
45                 ret.insert(ret.end(),
46                            it->second.first.begin(),
47                            it->second.first.end());
48             }
49         }
50         return ret;
51     }
52 };
```

Konstruktor klase `grid` iterira po vektoru S i svaku točku dodaje u vektor pripadajućeg kvadrata u δ-mreži (u kôdu je δ zamijenjeno s `d`). `operator[]` prima točku i vraća `bool` varijablu pridruženu kvadratu u kojem se ta točka nalazi (`bool` varijabla je `false` ako je kvadrat već provjeren). `operator()` prima točku i vraća vektor koji sadrži sve točke iz kvadrata u kojem se ona nalazi i susjednih kvadrata (samo onih koji nisu već provjereni).

Sada možemo predstaviti implementaciju Rabinovog algoritma.

```
53 result Rabin(const vector<point>& S) {
54     if(S.size() < 4) return brute_force(S, 0, S.size());
55     size_t sample_size = ceil(sqrt(S.size()));
56     uniform_int_distribution<size_t> dist(0, S.size() - 1);
57     vector<bool> T(S.size(), true);
58     vector<point> R;
59     while(R.size() < sample_size) {
60         size_t k = dist(randgen);
61         if(T[k]) {
62             R.push_back(S[k]);
63             T[k] = false;
64         }
65     }
66     result p(brute_force(R, 0, sample_size));
67     if(p.d == 0.0) return p;
68     grid G(S, sqrt(p.d));
69     for(auto& x : S) {
70         if(G[x]) {
71             vector<point> temp(G(x));
72             if(temp.size() > 1) {
73                 result r(brute_force(temp, 0, temp.size()));
74                 if(r.d < p.d) p = r;
75                 G[x] = false;
76             }
77         }
78     }
79     return p;
80 }
```

Objasnimo izbor `hash` funkcije. Funkcija mora preslikati varijablu tipa `gridindex` u jednu `size_t` vrijednost.[3] Pošto je `gridindex` par 64-bitnih cjelobrojnih brojeva s predznakom, najprije preslikavamo oba broja u nenegativne formulom

$$n \mapsto \begin{cases} 2n, & n \geq 0 \\ -2n-1, & n < 0 \end{cases}.$$

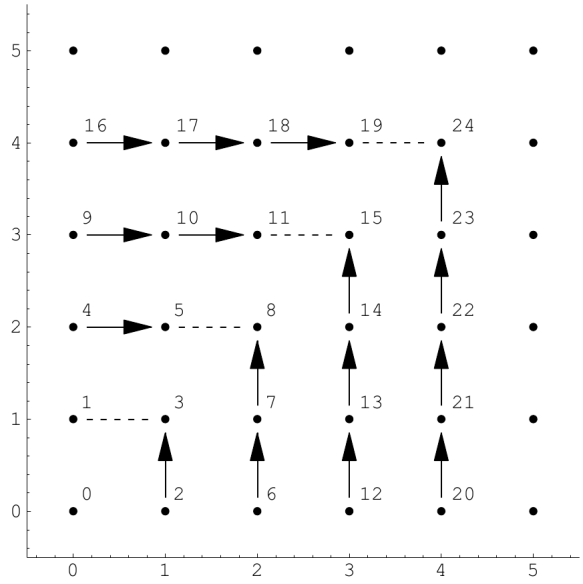
Uočimo da ovdje može doći do prelijevanja (*overflow*) brojeva u negativne. Međutim, pošto 64-bitni cijeli brojevi s predznakom mogu prikazati brojeve od -2^{63} do $2^{63}-1$, to će se dogoditi samo za brojeve koji su manji od -2^{62} ili veći od $2^{62}-1$. To nije problem, pošto će se prebacivanjem u `size_t` sve vrijednosti manje od 0 prebaciti u brojeve između 0 i $2^{8*\texttt{sizeof(size\_t)}}-1$, ali želimo osigurati da, pri prebacivanju u `size_t`, brojevi blizu nuli (uključujući negativne) ostanu blizu nuli.

[3] `size_t` je cjelobrojni tip bez predznaka. Veličina tipa (u smislu memorije) ovisi o sistemu – najčešće je 4 byte-a na 32-bitnim sistemima, a 8 byte-ova na 64-bitnim sistemima.

Zatim koristimo funkciju uparivanja[4] opisanu u [6], koja je definirana na sljedeći način:

$$(n, m) \mapsto \begin{cases} n^2 + n + m, & n \geq m \\ n + m^2, & n < m \end{cases} .$$

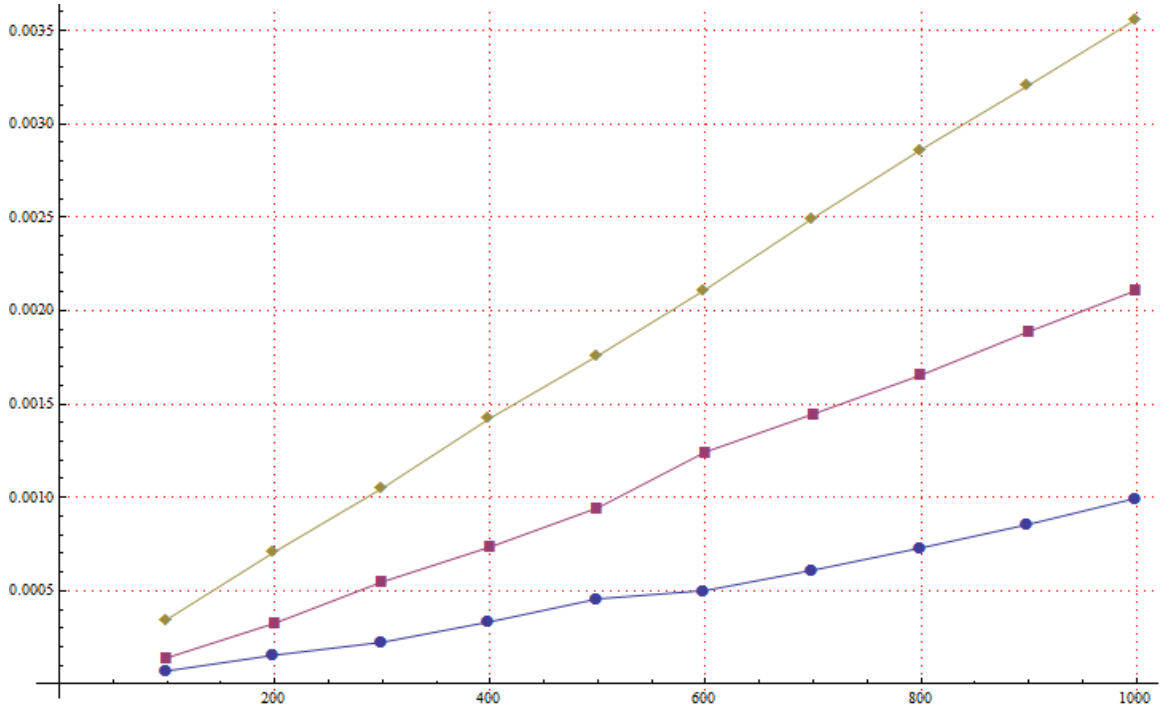
Preslikavanje ostvareno na ovaj način je grafički prikazano na sljedećoj slici:



Primijetimo da ovdje ponovno može doći do prelijevanja. Međutim, ako su `size_t` k-bitne vrijednosti, tada za najveću moguću vrijednost n i m, $2^k - 1$, vrijednost funkcije je $(2^k - 1)^2 + (2^k - 1) + (2^k - 1) = 2^{2k} - 1$, što je točno najveći broj prikaziv $2k$-bitnom cjelobrojnom vrijednošću bez predznaka. Uočimo da različitih parova `size_t` vrijednosti ima točno 2^{2k}. Pošto će povratna vrijednost biti tipa `size_t` te će sadržavati k najmanje značajnih bitova rezultata, svaka od mogućih 2^k vrijednosti koje su prikazive tipom `size_t` će biti rezultat za 2^k različitih parova. Dakle, ovim smo preslikali parove `size_t` vrijednosti u jednu uz optimalan raspored kolizijā, a time i sve `gridindex` vrijednosti. Štoviše, ako je $k \leq 64$ te su sve vrijednosti u `gridindex` parovima između $-2^{\frac{k}{2}-1}$ i $2^{\frac{k}{2}-1} - 1$, kolizijā nema.

Sljedeći graf uspoređuje Rabinov algoritam s najboljim varijantama prethodna dva algoritma.

[4]Funkcija uparivanja (eng. *pairing function*) je funkcija koja dva prirodna broja bijektivno preslikava u jedan prirodan broj, odnosno bijekcija sa $\mathbb{N}^2$ na $\mathbb{N}$.



Slika 4.1: ● `Divide_and_Conquer`; ■ `D_and_C_2`; ◆ `Rabin`

Iako nismo detaljno analizirali složenost Rabinovog algoritma, po grafu izgleda da vrijeme izvršavanja algoritma raste linearno s obzirom na duljinu vektora točaka. Zbilja, Rabin je i dokazao da je očekivana asimptotska složenost algoritma $O(n)$, uz dovoljno dobru `hash` operaciju za indekse kvadratā (`gridindex`).

Međutim, također možemo vidjeti da je Rabinov algoritam sporiji od obaju dosad prikazanih deterministčkih algoritama. Dodatno testiranje je pokazalo da je to slučaj čak i za vektore duljine 10,000,000.

Još jedna zanimljiva činjenica o ovom algoritmu, koju su Steve Fortune i John Hopcroft dokazali u [4], je da barem dio smanjenja složenosti s $O(n \log n)$ na $O(n)$ u Rabinovom algoritmu proizlazi iz pretpostavke da, u modelu računanja koji koristimo, operacije dijeljenja i $\lfloor \cdot \rfloor$ na varijablama s pomičnom točkom možemo obavljati u konstantnom vremenu.

4.2 Dietzfelbinger-Hagerup-Katajainen-Penttonen algoritam

Sada ćemo proučiti još jednu varijantu nedeterminističkog algoritma za rješavanje problema najbližeg para. Ovaj algoritam je opisan u [3] pod imenom "randomized-closest-pair". Algoritam je dosta sličan Rabinovom, ali se od njega razlikuje u nekoliko bitnih detalja. Slijedi opis algoritma.

Za ulazni vektor točaka S duljine n odredimo veličinu slučajnog uzorka, označimo ju sa s, tako da za neku konstantu c, $0 < c < \frac{1}{2}$, bude

$$18n^{\frac{1}{2}+c} \leq s = O\left(\frac{n}{\log n}\right).$$

Nasumično odaberimo s točaka iz S te s R označimo vektor koji sadrži točke iz uzorka čiji indeksi u S su različiti. Algoritmom `D_and_C_2` odredimo najbliži par točaka u R te označimo njihovu međusobnu udaljenost s δ. Ako je $\delta = 0$, vratimo pronađeni par. Inače, definirajmo skupove G_1, G_2, G_3 i G_4 na sljedeći način:

$$\begin{aligned}
G_1 &:= \left\{ [i(2\delta), (i+1)(2\delta)\rangle \times [j(2\delta), (j+1)(2\delta)\rangle \;\middle|\; i, j \in \mathbb{Z} \right\} \\
G_2 &:= \left\{ [i(2\delta)+\delta, (i+1)(2\delta)+\delta\rangle \times [j(2\delta), (j+1)(2\delta)\rangle \;\middle|\; i, j \in \mathbb{Z} \right\} \\
G_3 &:= \left\{ [i(2\delta), (i+1)(2\delta)\rangle \times [j(2\delta)+\delta, (j+1)(2\delta)+\delta\rangle \;\middle|\; i, j \in \mathbb{Z} \right\} \\
G_4 &:= \left\{ [i(2\delta)+\delta, (i+1)(2\delta)+\delta\rangle \times [j(2\delta)+\delta, (j+1)(2\delta)+\delta\rangle \;\middle|\; i, j \in \mathbb{Z} \right\}
\end{aligned}$$

Drugim riječima, skupovi G_1, G_2, G_3 i G_4 su 2δ-mreže, s tim da je G_2 translatirana za δ u smjeru x-osi, G_3 u smjeru y-osi te G_4 u smjeru x- i y-osi.

Za svaku točku u S odredimo u kojem se $2\delta \times 2\delta$ kvadratu u svakoj od ovih 2δ-mrežā nalazi. Zatim, za svaki kvadrat u kojem se nalaze barem dvije točke iz S algoritmom `brute_force` odredimo najbliži par u skupu svih točaka koje se nalaze u tom kvadratu. Najbliži od svih parova koje pronađemo ovim postupkom je najbliži par točaka u S.

U [3] se također može pronaći dokaz da je vrijeme izvršavanja ovog algoritma za n točaka $O(n)$ s vjerojatnošću barem $1 - 2^{-n^c}$. Pošto je $c > 0$, vrijedi

$$\lim_{n\to\infty}(1 - 2^{-n^c}) = 1 - 2^{\lim_{n\to\infty} -n^c} = 1 - 2^{\lim_{n\to-\infty} n} = 1.$$

Kako bismo implementirali ovaj algoritam, trebamo malo modificirati funkciju `gridbin` i klasu `grid` koje smo koristili za implementaciju Rabinovog algoritma. Moramo omogućiti translatiranje mrežā, pa ćemo `gridbin` redefinirati na sljedeći način:

```
inline gridindex gridbin(const point& p, const double& d,
                         const double& dx, const double& dy) {
    return make_pair(floor((p.first + dx)/d), floor((p.second + dy)/d));
}
```

Klasu `grid` modificiramo da koristi novu `gridbin` funkciju. Također, pri dobavljanju vektora točaka u funkciji `operator()` više ne moramo vraćati i točke iz susjednih kvadratā.

```
class grid {
    unordered_map<gridindex, pair<vector<point>, bool>> data;
    double d, dx, dy;

public:
    grid(const vector<point>& S, const double& _d,
         const double& _dx, const double& _dy)
        : d(_d), dx(_dx), dy(_dy) {
        for(auto& p : S)
            data[gridbin(p, d, dx, dy)].first.push_back(p);
        for(auto& p : data) p.second.second = true;
    }

    bool& operator[] (const point& p) {
        return (data.at(gridbin(p, d, dx, dy))).second;
    }

    vector<point>& operator() (const point& p) {
        return (data.at(gridbin(p, d, dx, dy))).first;
    }
};
```

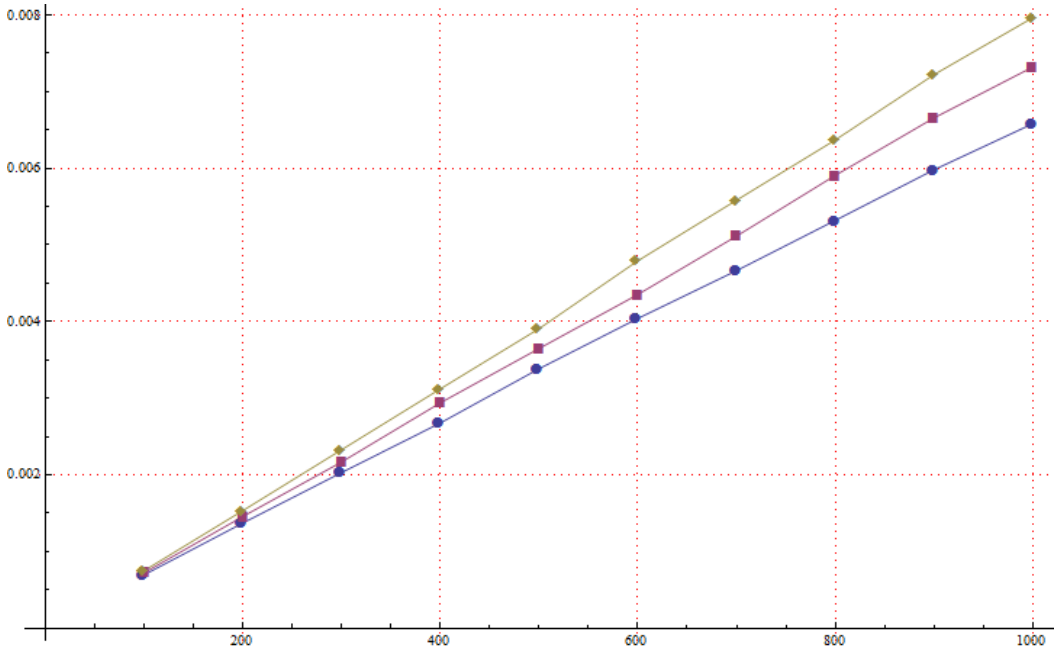
Sljedeću funkciju ćemo koristiti da odredimo veličinu uzorka za danu duljinu ulaznog vektora.

```
inline size_t getsamplesize(const vector<point>& S,
                            const double& c,
                            const double& k) {
    double n = static_cast<double>(S.size()),
        low = 18.0 * pow(n, 0.5 + c),
        high = k * n / log(n);
    if(low >= high) return (ceil(low));
    return ceil(high);
}
```

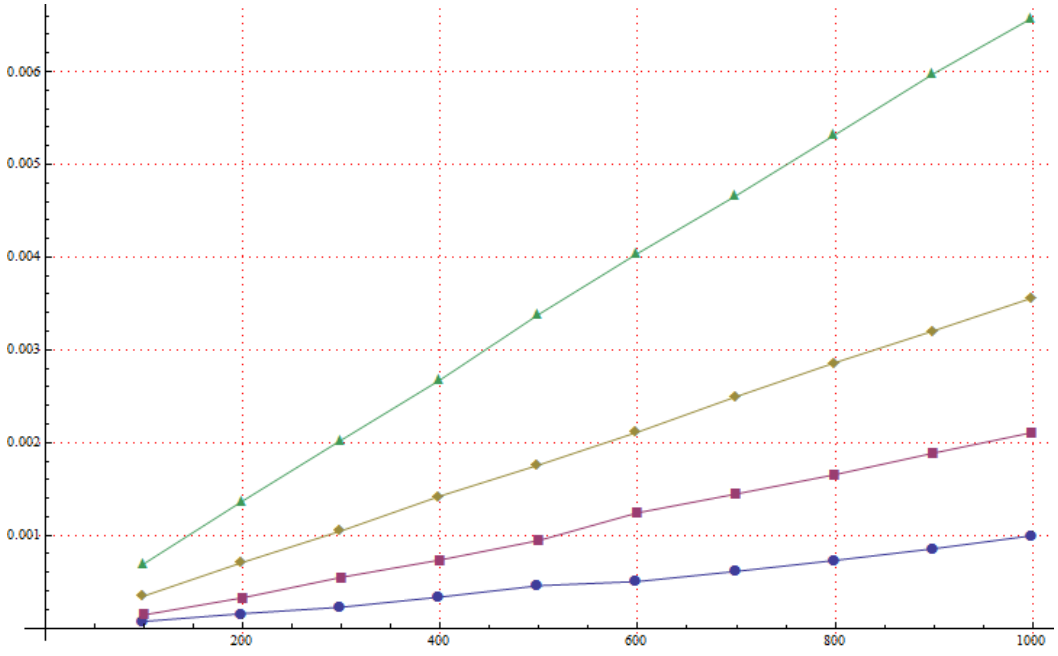
Konačno, prikažimo implementaciju algoritma.

```
46 result DHKP(const vector<point>& S,
47              const double& c, const double& k) {
48     size_t s(getsamplesize(S, c, k));
49     uniform_int_distribution<size_t> dist(0, S.size()-1);
50     vector<bool> T(S.size(), true);
51     vector<point> R;
52     for(size_t i = 0; i < s; ++i) {
53         size_t k = dist(randgen);
54         if(T[k]) {
55             R.push_back(S[k]);
56             T[k] = false;
57         }
58     }
59     vector<point> R2(R);
60     sort(R.begin(), R.end(), compare_x);
61     sort(R2.begin(), R2.end(), compare_y);
62     result p(D_and_C_2(R, R2, 0, R.size()));
63     if(p.d == 0.0) return p;
64     double delta = sqrt(p.d),
65         delta_2 = 2.0 * delta;
66     grid G[] = {grid(S, delta_2, 0.0, 0.0),
67                 grid(S, delta_2, delta, 0.0),
68                 grid(S, delta_2, 0.0, delta),
69                 grid(S, delta_2, delta, delta)};
70     for(auto& x : S) {
71         for(uint8_t i = 0; i < 4; ++i) {
72             if(G[i][x]) {
73                 vector<point> *temp = &(G[i](x));
74                 if((*temp).size() > 1) {
75                     result r(brute_force(*temp, 0, (*temp).size()));
76                     if(r.d < p.d) p = r;
77                     G[i][x] = false;
78                 }
79             }
80         }
81     }
82     return p;
83 }
```

Parametrom `c` biramo "sigurnost" linearne složenosti algoritma, a parametrom `k` biramo koliko je s (veličina slučajnog uzorka) blizu $O\left(\frac{n}{\log n}\right)$. Primijetimo da parametar `k` može biti bilo koji broj veći ili jednak od 0, pošto za $0 < c < \frac{1}{2}$ vrijedi $18n^{\frac{1}{2}+c} = O\left(\frac{n}{\log n}\right)$. Stoga, fiksirat ćemo vrijednost parametra `k` na 0 te ćemo proučiti utjecaj izbora parametra `c` na brzinu algoritma. Idući graf uspoređuje izmjerena vremena za različite vrijednosti parametra `c`.



Slika 4.2: ● c = 0.001; ■ c = 0.1; ◆ c = 0.2



Slika 4.3: ● `Divide_and_Conquer`; ■ `D_and_C_2`; ◆ `Rabin`; ▲ `DHKP`

Kao što je i bilo za očekivati, povećavanjem parametra c se brzina algoritma smanjuje. No, također možemo primijetiti da već za vrlo male vrijednosti parametra c dobivamo praktički linearnu složenost algoritma.

Međutim, kao što možemo vidjeti iz drugog grafa koji uspoređuje algoritam DHKP uz c = 0.001 i k = 0 s prijašnjim algoritmima, ovaj algoritam je sporiji od svih dosad proučenih algoritama. Kao i dosad, algoritam je testiran i na vektorima duljine 10,000,000 te je i u tom slučaju sporiji čak i od Rabinovog algoritma.

Poglavlje 5

Heuristički pristup

Za kraj, opisat ćemo pokušaj dobivanja približnog rješenja problema najbližeg para meta-heuristikom simuliranog taljenja. Zbog jednostavnosti izvedbe algoritma, potrebna nam je sljedeća dodatna definicija tipa i jedne funkcije:

```
1  #define indexpair pair<size_t, size_t>
2
3  inline double distance(const vector<point>& S,
4                         const indexpair& x) {
5      return distance(S[x.first], S[x.second]);
6  }
```

Algoritam počinje odabirom slučajnog para točaka. Za to koristimo sljedeću funkciju:

```
7  inline indexpair pickrandom(uniform_int_distribution<size_t>& distr) {
8      indexpair ret(make_pair(distr(randgen), distr(randgen)));
9      while(ret.first == ret.second) ret.first = distr(randgen);
10     return ret;
11 }
```

Nakon toga, potrebna nam je funkcija koja za dani par točaka vraća "susjedan" par, pošto tako izvodimo jedan korak simuliranog taljenja. Sljedeća funkcija za dani par točaka (a, b) odabere nasumičnu točku c različitu od a i b te vrati (a, c) ako je točka c bliža točki a nego točki b, a (c, b) inače.

```
12 inline indexpair neighbour(const vector<point>& S,
13                            const indexpair& x,
14                            uniform_int_distribution<size_t>& distr) {
15     size_t i = distr(randgen);
16     while(i == x.first || i == x.second) i = distr(randgen);
17     if(distance(S[x.first], S[i]) < distance(S[x.second], S[i]))
18         return make_pair(x.first, i);
19     return make_pair(i, x.second);
20 }
```

Sada definiramo funkciju koja određuje hoće li novi par točaka biti prihvaćen s obzirom na udaljenosti prošlog i novog para točaka te trenutnu temperaturu.

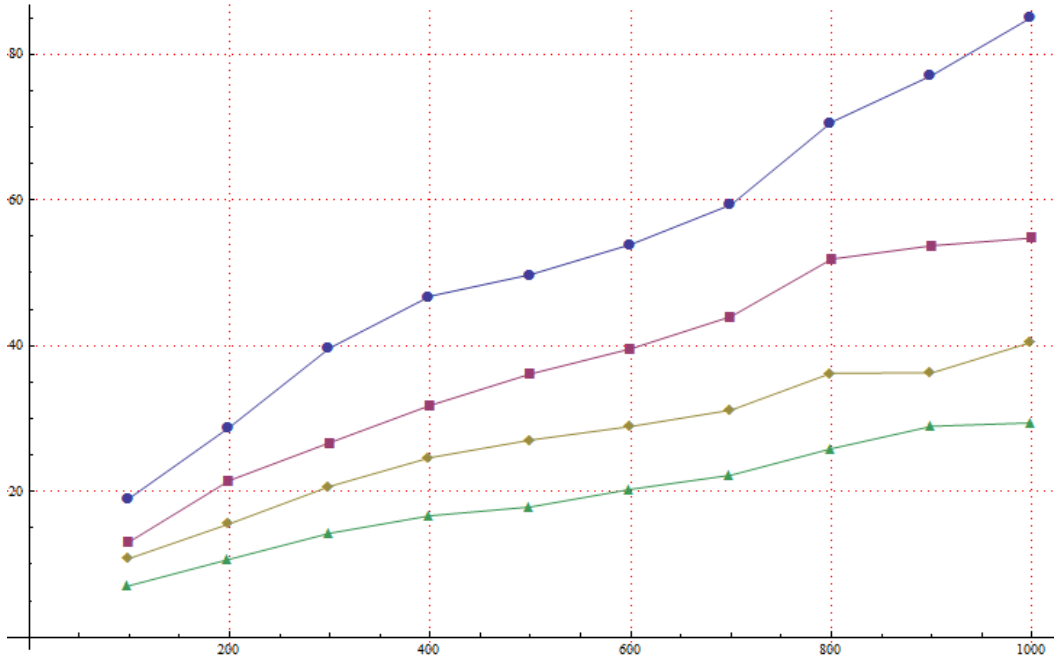
```
21 inline bool accept(const double& d_old,
22                    const double& d_new,
23                    const double& T) {
24     if(d_new < d_old) return true;
25     static uniform_real_distribution<double> distr(0.0, 1.0);
26     return (distr(randgen) < exp((d_old - d_new)/T));
27 }
```

Slijedi implementacija algoritma.

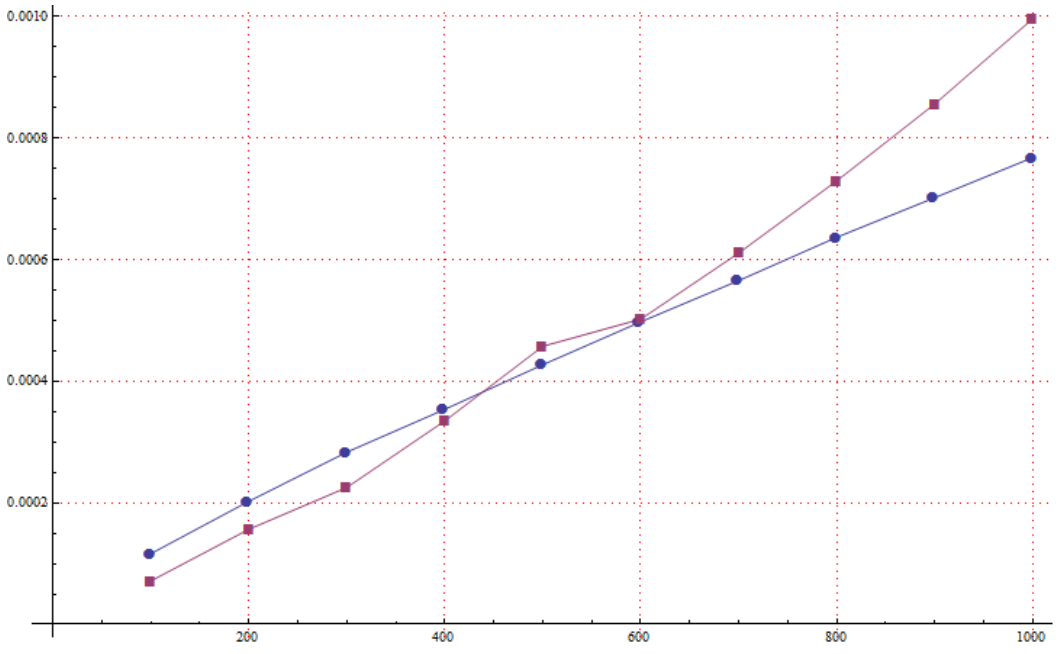
```
28 result heuristic_cp(const vector<point>& S,
29                      const double& c) {
30     if(S.size() < 4) return brute_force(S, 0, S.size());
31     uniform_int_distribution<size_t> distr(0, S.size()-1);
32     double n = static_cast<double>(S.size()),
33         T = 4294967296.0,
34         divisor = pow(T, log2(n) / (c * n));
35     indexpair current = pickrandom(distr),
36         best(make_pair(0, 1));
37     double d_current = distance(S, current),
38         d_best = distance(S, best);
39     while(T > 1.0) {
40         indexpair newr(neighbour(S, current, distr));
41         double d_new = distance(S, newr);
42         if(accept(d_current, d_new, T)) {
43             current = newr;
44             d_current = d_new;
45             if(d_current < d_best) {
46                 best = current;
47                 d_best = d_current;
48             }
49         }
50         T /= divisor;
51     }
52     result ret(S[best.first], S[best.second], d_best);
53     return ret;
54 }
```

Označimo početnu vrijednost varijable T s T_0. Pošto u svakom koraku algoritma varijablu T dijelimo s $T_0^{\frac{\log_2 n}{cn}}$ te zaustavljamo algoritam kad vrijednost T postane 1, algoritam će odraditi $\frac{cn}{\log_2 n} + 1$ korakā, što znači da mu je složenost $O\left(\frac{n}{\log n}\right)$.

Sljedeći graf prikazuje prosječnu relativnu grešku za tisuću poziva algoritma na vektorima duljinā od 100 do 1000 za različite vrijednosti parametra c.



Slika 5.1: ● c = 1; ■ c = 2; ◆ c = 4; ▲ c = 8



Slika 5.2: ● `heuristic_cp` (c = 16); ■ `Divide_and_Conquer`

Relativna graška je izračunata dijeljenjem udaljenosti para točaka kojeg je vratio algoritam `heuristic_cp` udaljenošću stvarnog najbližeg para točaka.

Pošto vrijeme izvršavanja algoritma očito linearno ovisi o parametru `c`, u drugom grafu je uspoređeno vrijeme izvršavanja algoritma za `c` = 16 s vremenom izvršavanja algoritma `Divide_and_Conquer`. Možemo primijetiti da je za tu vrijednost parametra `c` algoritam `heuristic_cp` brži od `Divide_and_Conquer` za vektore duljine veće od ~450.

Bibliografija

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest i C. Stein, *Introduction to Algorithms*, The MIT Press, 2009.

[2] S. Dasgupta, S. H. Papadimitrou i U. V. Vazirani, *Algorithms*, 2006.

[3] M. Dietzfelbinger, T. Hagerup, J. Katajainen i M. Penttonen, *A Reliable Randomized Algorithm for the Closest-Pair Problem*, Journal of Algorithms **25** (1997), 19–51.

[4] S. Fortune i J. Hopcroft, *A Note on Rabin's Nearest-Neighbor Algorithm*, Cornell University Computer Science Technical Reports (1978).

[5] M. O. Rabin, *Probabilistic algorithms*, Algorithms and Complexity: New Directions and Recent Results (1976), 21–39.

[6] M. Szudzik, *An Elegant Pairing Function*, Wolfram Research, 2006, `http://www.szudzik.com/ElegantPairing.pdf`.

Sažetak

U ovom radu smo proučili šest algoritama za rješavanje problema najbližeg para u $\mathbb{R}^2$. Pokazali smo kako se problem može riješiti determinističkim algoritmom složenosti $O(n \log n)$, a nedeterminističkim algoritmima složenost možemo spustiti do $O(n)$.

Ipak, testiranjem smo utvrdili da je na praktičnim veličinama ulaznih podataka od egzaktnih algoritama najefikasniji onaj koji je predstavljen u odjeljku 3.1 pod imenom `Divide_and_Conquer`, čija složenost je $O(n \log^2 n)$.

Na kraju smo predstavili i jedan parametrizirani heuristički algoritam složenosti $O\left(\frac{n}{\log n}\right)$ koji pronalazi približno rješenje, čija točnost se može povećati nauštrb vremena izvođenja.

Summary

In this paper we have examined six different algorithms for solving the closest pair of points problem in $\mathbb{R}^2$. We have shown that the problem can be solved with deterministic algorithms in $O(n \log n)$ time and in $O(n)$ time with non-deterministic algorithms.

However, testing these algorithms has revealed that, for practical input sizes, the most efficient exact algorithm is the one we have labelled `Divide_and_Conquer` which was presented in section 3.1 and which runs in $O(n \log^2 n)$ time.

In the final chapter we have presented a parameterised simulated annealing algorithm which finds an approximate solution in $O\left(\frac{n}{\log n}\right)$ time, accuracy of which can be increased at the expense of speed.

Životopis

Ivo Doko rođen je 8. kolovoza 1987. u Makarskoj. Osnovnoškolsko obrazovanje stekao je u Brelima, a opću gimnaziju pohađao je u Srednjoj školi fra Andrije Kačića Miošića u Makarskoj, gdje je i maturirao 2006. godine.

2006. godine upisuje preddiplomski studij matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Titulu sveučilišnog prvostupnika matematike stekao je 2011. godine te iste godine upisuje diplomski studij računarstva i matematike kojeg je ovo diplomski rad.

Tokom studija sudjelovao je na Intelovom natjecanju *Accelerate Your Code*[1] na kojem je, u timu s kolegom Ivom Ivaniševićem, bio plasiran na 45. mjesto u konkurenciji više od 500 timova. Redovito je sudjelovao i na timskom internetskom natjecanju *Internet Problem Solving Contest (IPSC)*[2] u timu s kolegama s Matematičkog odsjeka PMF-a i Max Planck instituta za softverske sustave (SR Njemačka) gdje je najbolji plasman ostvario 2013. godine osvojivši 256. mjesto s 12 bodova.

[1] `https://software.intel.com/en-us/articles/AYC-early2012_home`
[2] `http://ipsc.ksp.sk/`